Ramesh

Q: A Developer’s Guide

Version 0.1 June 12, 2017

Contents

Q: A Developer's Guide

1 Introduction

The aim of this document is to provide a step-by-step guide to being a library developeri, working on the internals of Q. We will also discuss how packaging and installation are done and how to operate as a Q developer. A Q developer need have **no** idea about the internals of Q — it is just another Lua package. Of course, understanding the internals, to some extent, allows for more efficient usage of the Q primitives.

2 Getting Started

2.1 Building your machine for the first time

Starting with a minimal Ubuntu install, you should execute Indrajeet $\boxed{\textbf{TO BE COMPLETED}}$

2.2 Environment Variables

So, you want to modify the guts of Q? Here's a step by step guide.

1. Say, you are in `/home/subramon/WORK/`
2. `git clone https://github.com/NerdWalletOSS/Q.git`
3. Set the following environment variables using `source setup.sh -f`. Note that this is just a convenience. If you want, you can set them yourself but then the onus is on you to get things right. These are
   (a) `QC_FLAGS` — these will be used as flags to gcc when creating `.o` files
   (b) `Q_LINK_FLAGS` — these will be used as flags to gcc when creating `.so` files

(c) `Q_ROOT` — This is where artifacts created by the build provess will be stored. As of now, they are
   i. `Q_ROOT/lib/` — contains `libq_core.so`
   ii. `Q_ROOT/include/` — contains `q_core.h`
   iii. `Q_ROOT/tmpl/` — contains templates, used for dynamic code generation

(d) `Q_DATA_DIR` — This is where data files will be stored. Think of this as a tablespace and keep a separate one for each project you are working

(e) `Q_METADATA_DIR` — This is where meta data files will be stored. Think of this as a tablespace and keep a separate one for each project you are working on. Default will be `Q_ROOT/meta/`

(f) `LD_LIBRARY_PATH` Make sure that this includes `Q_ROOT/lib/` This is where `libq_core.so` will be created

(g) `LUA_PATH`, Section 2.4

2.2.1 Consequences

There are some important consequences of the above.

1. **Do not set** these environment variables in any of your scripts.

2. **Do not use** `Q_ROOT` anywhere except in `Q/UTILS/build`

3. In your Lua scripts, you must specify the entire path of the file you want to require e.g.,

```
local foo = require 'Q/UTILS/lua/foo'
```

2.3 Building

C programs are used to augment Lua in two important ways

1. to help with code generation and to perform some functionality that could not be done easily (or in a performant manner) in Lua. Examples of these are text converters like `txt_to_I8` or `get_cell`

2. the computational workhorse. This is where the heavy lifting happens.

You will note a bit of a circular dependency. We need C code to create C code. This is broken in one of two ways

1. Execute `Q/UTILS/mk_core_so.lua` This creates the following files

   (a) `Q_ROOT/include/q_core.h` — which is used for `ffi.cdef()`

   (b) `Q_ROOT/lib/q_core.so` — which is used for `ffi.load()`

   You are have the C functionality needed for code generation

2. Within `Q/UTILS/build/`, do `make clean; make`

2.4 Masquerading as a Q developer

From time to time, you will need to pretend to be a Q developer so that you can test your code. To enable this to happen without re-installing Q, you set `LUA_PATH` as below. Note the double semi-colon at the end. That is needed. Srinath to fill in the gaps $\boxed{\textbf{TO BE COMPLETED}}$

```
/home/subramon/WORK/?.lua;;+
```

2.5 Installation

At some point in the not too distant future, Q will be installed as

```
sudo luarocks install Q
```

Until then, it is installed as

```
cd $Q_SRC_ROOT; sudo bash q_install.sh
```

The Q developer does not need to set any of the environment variables of Section 2.2 nor do they need to set `LUA_INIT` A sample Q script looks as follows

```
Q = require 'Q'
x = Q.mk_col({10, 20, 30, 40}, 'I4')
Q.print_csv(x, nil, "")
os.exit()
```

3 Lua Coding Conventions

This section deals with coding conventions to be followed by a library developer writing Lua code.

1. Do **not** pollute the global name space. So, all variables are **local**.

2. Do not use `dofile`. Use `require instead`

3. LuaJIT scripts that invoke pthreads (indirectly) should end with `os.exit()`

1 LuaJIT and OpenMP

Ciprian Tomoiaga wrote: *I have a C function containing OpenMP clause which I call with ffi. On my machine, the program terminates with a segmentation fault. However, it works without a problem on a Mac and on another machine.*

Mike Pall responded: The most likely cause is the dreaded pthread issue: the main executable must be compiled/linked with -pthread (note: this is not the same as -lpthread!). Otherwise a shared library which is loaded later on and uses threads may cause a crash. The easiest way to test this hypothesis is to rebuild LuaJIT with:

```
make TARGET_FLAGS=-pthread
```

and then try again.

Ciprian Tomoiaga wrote: *I recompiled with the specified opition, but it still crashes. :(Do I have any other options?*

Mike Pall responded: Well I tried myself and it prints two lines, but then crashes when the shared library is unloaded (as a consequence of `lua_close()`). Looks like OpenMP doesn't like to be unloaded — this never happens with programs that are statically linked against it.